

CHALLENGE

40. Create a program to find the **sum and average** of all elements in an array.
41. Create a program to find **number of occurrences** of an element in an array.
42. Create a program to find the **maximum and minimum element** in an array.
43. Create a program to **check** if the given array is **sorted**.
44. Create a program to return a new array **deleting** a specific element.
45. Create a program to **reverse** an array.
46. Create a program to check is the array is **palindrome** or not.
47. Create a program to **merge two sorted arrays**.
48. Create a program to **search** an element in a **2-D array**.
49. Create a program to do **sum and average** of all elements in a **2-D array**.
50. Create a program to find the sum of two **diagonal elements**.





Revision

1. Comments
2. While Loop
3. Methods
4. Return statement
5. Arguments
6. Arrays
7. 2D Arrays



Practice Exercise

While Loop, Methods & Arrays

Answer in True/False:

1. A 'while' loop in Java continues to execute as long as its **condition** is **true**.
2. The body of a 'while' loop will execute **at least once**, regardless of the condition.
3. A 'while' loop **cannot be used** for iterating over an array.
4. Infinite loops are **not possible** with 'while' loops in Java.
5. A method in Java can **return more than one value** at a time.
6. It's **mandatory** for every Java method to have a **return type**.
7. The **size** of an array in Java can be **modified** after it is created.
8. Arrays in Java can contain elements of **different data types**.
9. An **array** is a **reference type** in Java.
10. Java arrays are **zero-indexed**, meaning the first element has an index of 0.





Practice Exercise

While Loop, Methods & Arrays

Answer in True/False:

1. A 'while' loop in Java continues to execute as long as its condition is true. True
2. The body of a 'while' loop will execute at least once, regardless of the condition. False
3. A 'while' loop cannot be used for iterating over an array. False
4. Infinite loops are not possible with 'while' loops in Java. False
5. A method in Java can return more than one value at a time. False
6. It's mandatory for every Java method to have a return type. True
7. The size of an array in Java can be modified after it is created. False
8. Arrays in Java can contain elements of different data types. False
9. An array is a reference type in Java. True
10. Java arrays are zero-indexed, meaning the first element has an index of 0. True



KG Coding

Some Other One shot Video Links:

- [Complete HTML](#)
- [Complete CSS](#)
- [Complete JavaScript](#)
- [Complete React and Redux](#)
- [One shot University Exam Series](#)



<http://www.kgcoding.in/>

Our  YouTube Channels

KG Coding Android App



[KG Coding](#)



[Knowledge GATE](#)



[KG Placement Prep](#)



[Sanchit Socket](#)



6 Classes & Objects

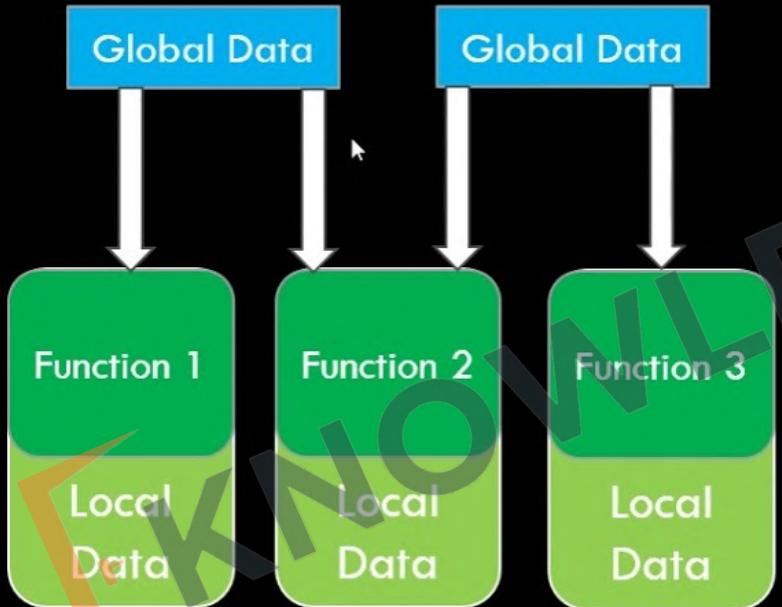
1. Process vs Object Oriented
2. Instance Variables and Methods
3. Declaring and Using Objects
4. Class vs Object
5. This & Static Keyword
6. Constructors & Code Blocks
7. Stack vs Heap Memory
8. Primitive vs Reference Types
9. Variable Scopes
10. Garbage Collection & Finalize



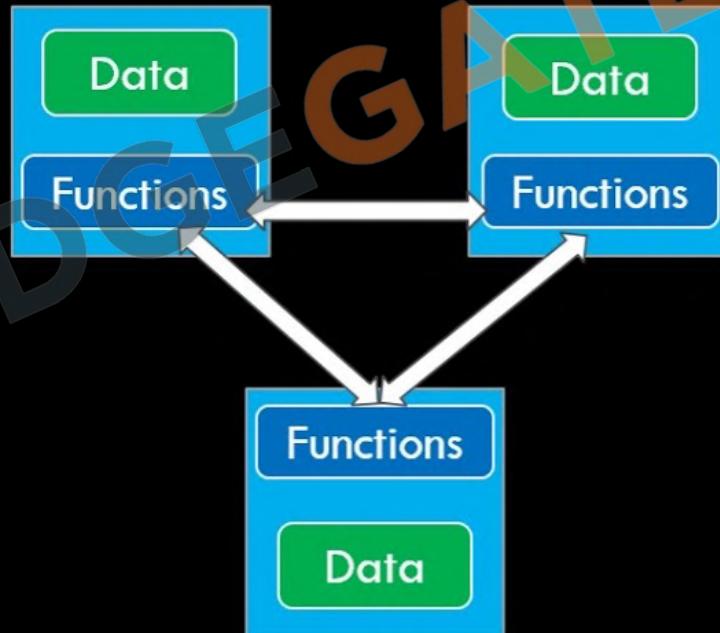


6.1 Process vs Object Oriented

Procedural Oriented Programming



Object Oriented Programming



6.1 Process vs Object Oriented

- Procedural



Withdraw, deposit, transfer

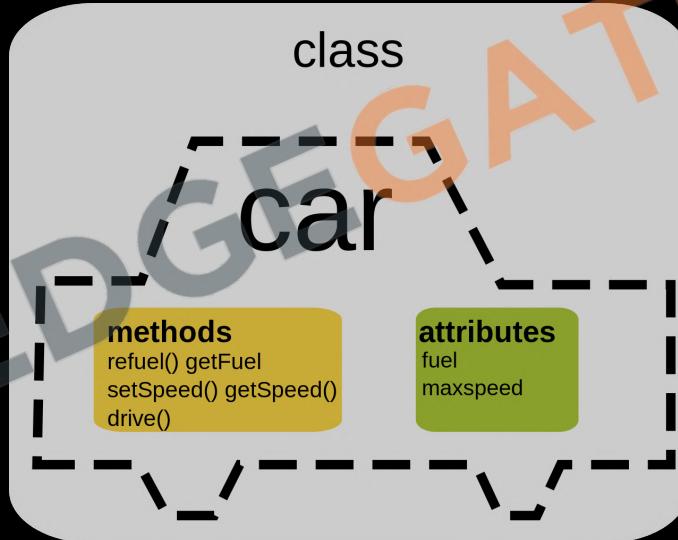
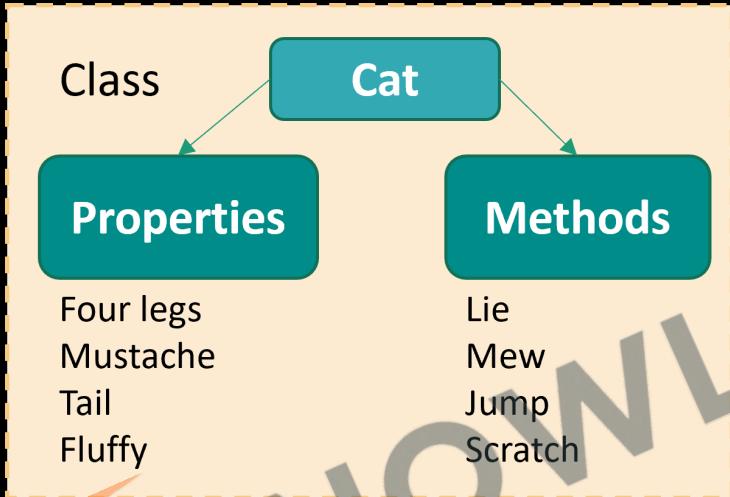
- Object Oriented



Customer, money, account



6.2 Instance Variables and Methods





6.3 Declaring Objects

STATEMENT

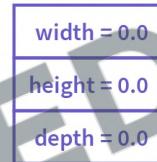
Box mybox ;

EFFECT

mybox



mybox = new Box



1. Object Creation: **new** instantiates a new object of a class.
2. Memory Allocation: Allocates memory for the object in the **heap**.
3. Constructor Invocation: Calls the class **constructor** to initialize the object.
4. Reference Return: Returns a **reference** to the created object.
5. Array Creation: Also used for creating arrays, like `int[] arr = new int[5];`.
6. Dynamic Allocation: Unlike static allocation, **new** allows for **dynamic memory allocation**, allocating memory at runtime.



6.3 Declaring Objects Syntax

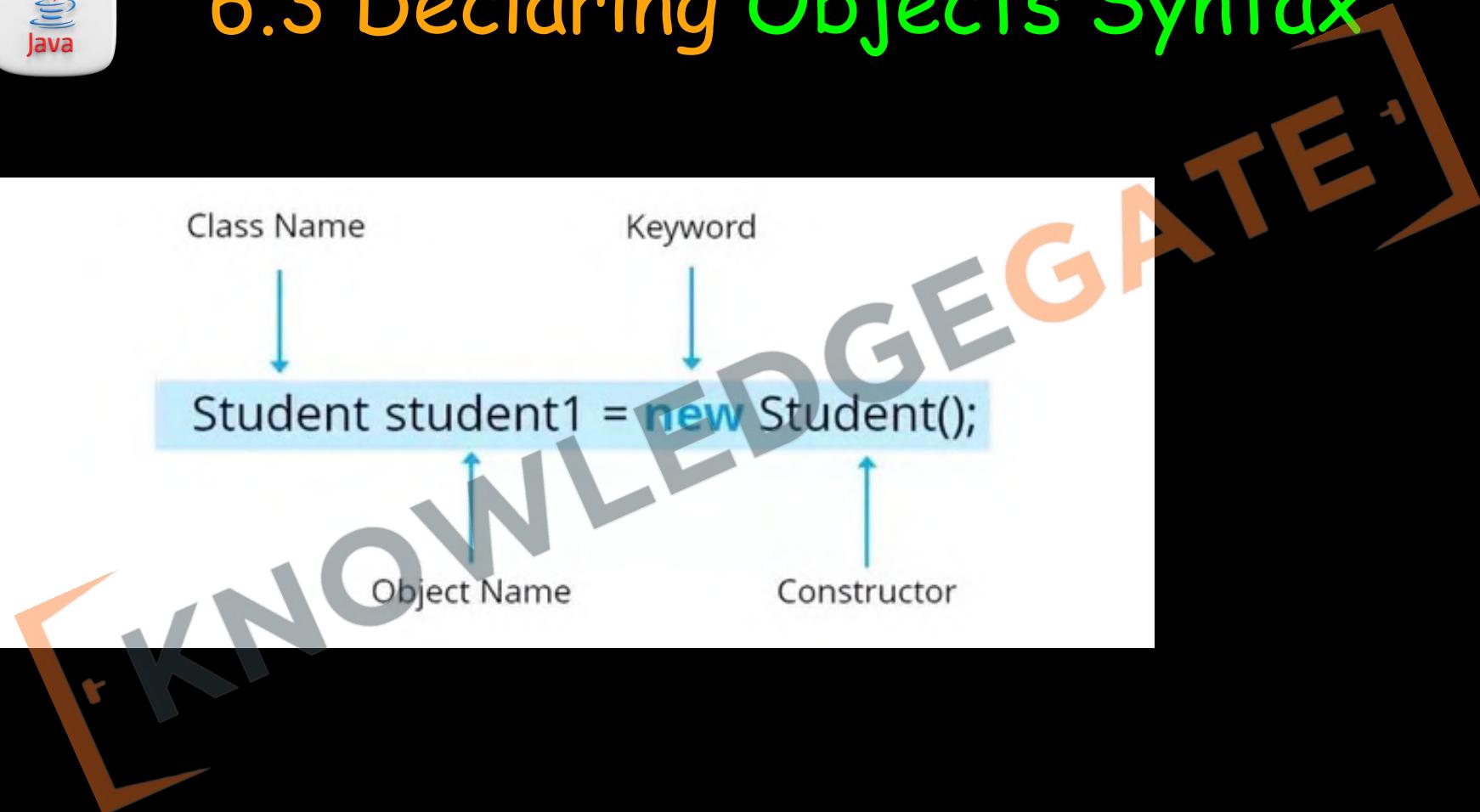
Class Name

Keyword

```
Student student1 = new Student();
```

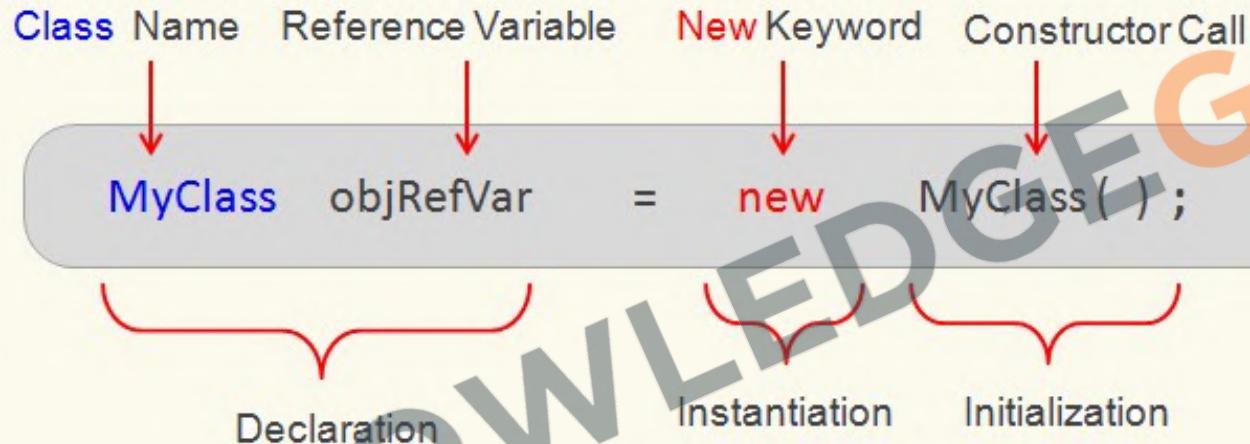
Object Name

Constructor





6.3 Declaring Objects Syntax



objRefVar is a Reference Variable of MyClass Type .

new is a Java Keyword used To Instantiate a Class .



6.3 Using Objects



Access properties using . Operator like `product.price`



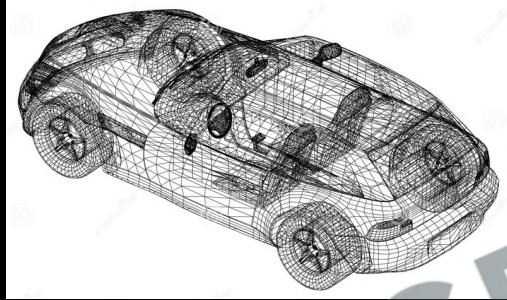
6.4 Class vs Object



Class is a blueprint; Objects are real values in memory.



6.4 Class vs Object

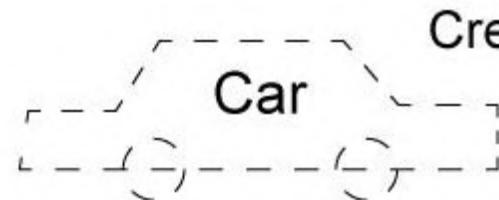


Class is a **blueprint**; Objects are **real values in memory**.



6.4 Class vs Object

Class



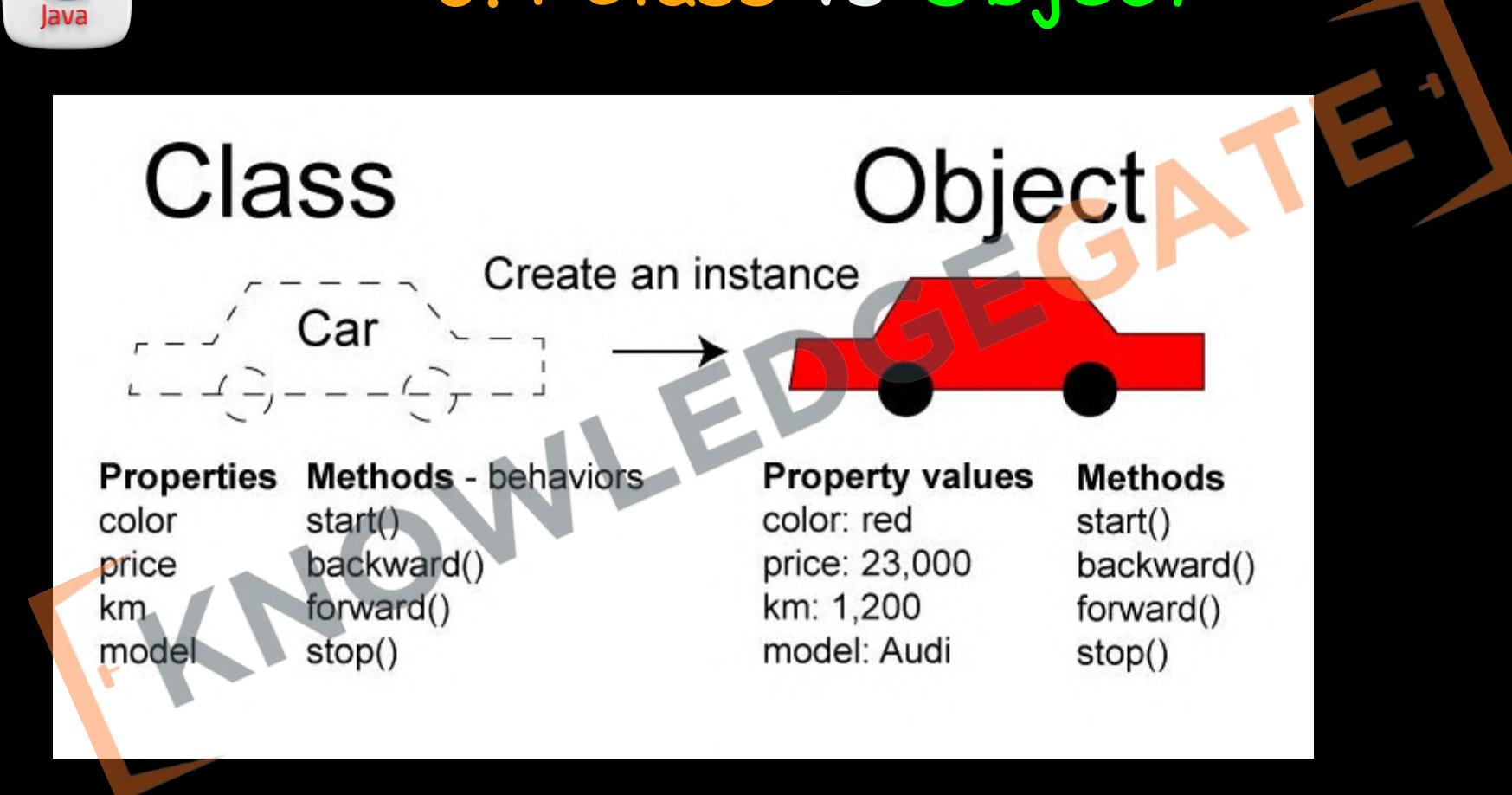
Create an instance

Object



Properties	Methods - behaviors
color	start()
price	backward()
km	forward()
model	stop()

Property values	Methods
color: red	start()
price: 23,000	backward()
km: 1,200	forward()
model: Audi	stop()



6.5 This Keyword

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

1

`this` can be used to refer current class instance variable.

2

`this` can be used to invoke current class method (implicitly)

3

`this()` can be used to invoke current class constructor.

4

`this` can be passed as an argument in the method call.

5

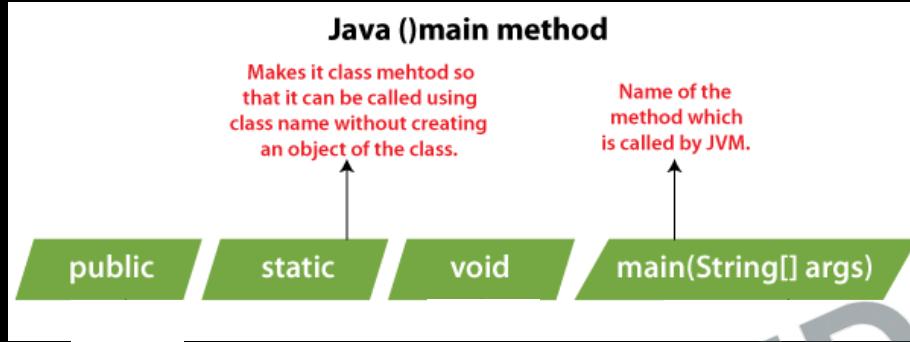
`this` can be passed as argument in the constructor call.

6

`this` can be used to return the current class instance from the method.

1. **Current Instance:** Refers to the **current class instance** variable.
2. **Constructor Call:** Can be used to invoke a **constructor of the same class** (`this()`).
3. **Method Call:** Invokes a **method of the current object**.
4. **Pass as Argument:** Can be passed as an **argument in the method call**.
5. **Return the Current Class Instance:** Can return the **current class instance** from the method.

6.5 Static Keyword



1. **Static Variables:** Belong to the class, not individual instances.
Shared among all instances of the class.
2. **Static Methods:** Can be called without creating an object of the class. Can only directly access static variables and other static methods.
3. **No Access to Non-static Members:** Static methods and blocks cannot directly access non-static members (variables and methods) of the class.



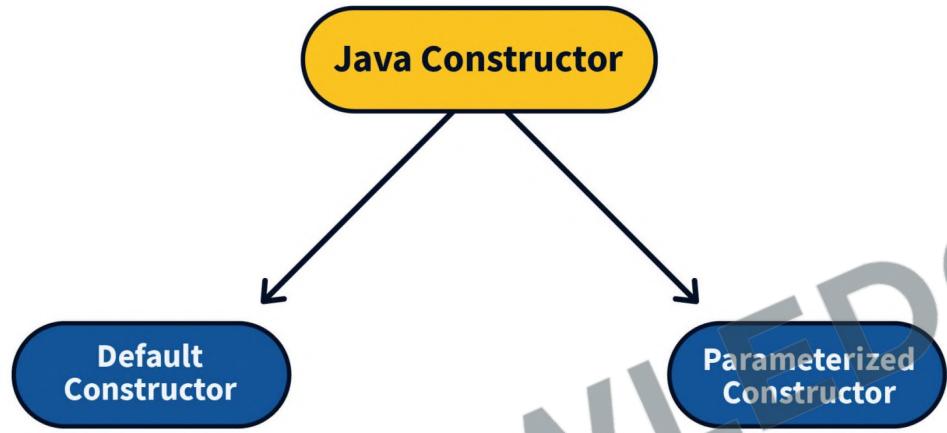
6.6 Constructors

```
public class Car {  
    1 usage  
    String color;  
    1 usage  
    float price;  
  
    1 usage  new *  
    Car() { // Default constructor  
        color = "Black";  
        price = 50000;  
    }  
  
    new *  
    public static void main(String[] args) {  
        Car swift = new Car();  
    }  
}
```

1. **Purpose:** Constructors **initialize new objects** and **set initial states** for the object's attributes.
2. **Naming:** A constructor must have the **same name as the class** in which it is declared.
3. **No Return Type:** Constructors **do not have a return type**, not even void.
4. **Automatic Calls:** A constructor is **automatically called** when an instance of a class is created.



6.6 Constructors (Types)



1. Default Constructor: If no constructor is explicitly defined, Java provides a default constructor that initializes all member variables to default values.
2. Parameterized Constructors: Constructors can have parameters to pass values when creating an object, allowing for different initializations.

```
Car(String carColor, float currPrice) {  
    color = carColor;  
    price = currPrice;  
}
```



6.6 Constructors (Chaining)

Constructor Chaining



```
student(){  
    this(5)  
}
```

```
student(){  
    this(id, "hi")  
}
```

```
student(int id,  
        String msg){  
    this(id, "hi")  
}
```

1. Within Same Class: Using `this()` to call **another constructor** in the same class.
2. First Statement: `this()` must be the **first statement** in a constructor.
3. No Loop: Constructor **chaining** **can't form a loop**; it must have a termination point.



Java

6.6 Code Blocks

```
{  
    System.out.println("This is a Initialization Block");  
    color = "Black";  
    price = 50000;  
}
```

```
static {  
    System.out.println("This is a Static Block");  
}
```

```
if (true) { // code block  
    System.out.println("Code Block");  
}
```

1. Scope: Code blocks {} determine the scope of variables.
2. Local Variables: Variables inside a block are not accessible outside it.
3. Initialization Block: Blocks without static run each time an instance is created.
4. Static Block: Blocks with static run once when the class is loaded.

CHALLENGE

51. Create a **Book** class for a library system.

- Instance variables: `title`, `author`, `isbn`.
- Static variable: `totalBooks`, a counter for the total number of book instances.
- Instance methods: `borrowBook()`, `returnBook()`.
- Static method: `getTotalBooks()`, to get the total number of books in the library.

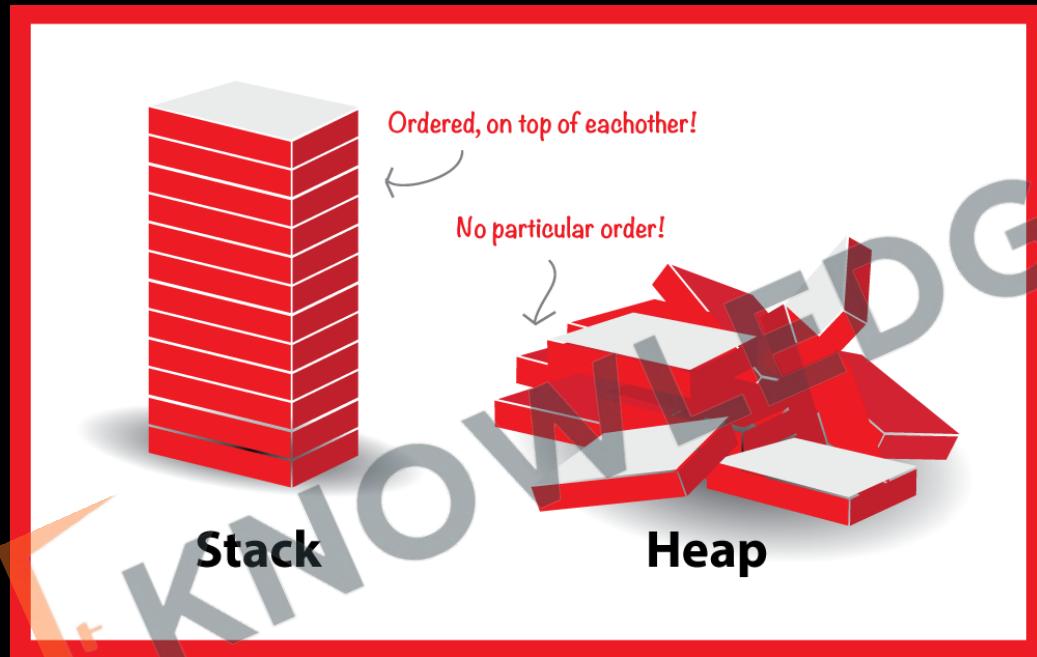
52. Design a **Course** class.

- Instance variables: `courseName`, `enrolledStudents`.
- Static variable: `maxCapacity`, the maximum number of students for any course.
- Instance methods: `enrollStudent(String studentName)`, `unenrollStudent(String studentName)`.
- Static method: `setMaxCapacity(int capacity)`, to set the maximum capacity for courses.





6.7 Stack vs Heap Memory

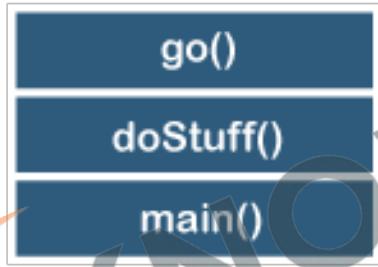




6.7 Stack vs Heap Memory

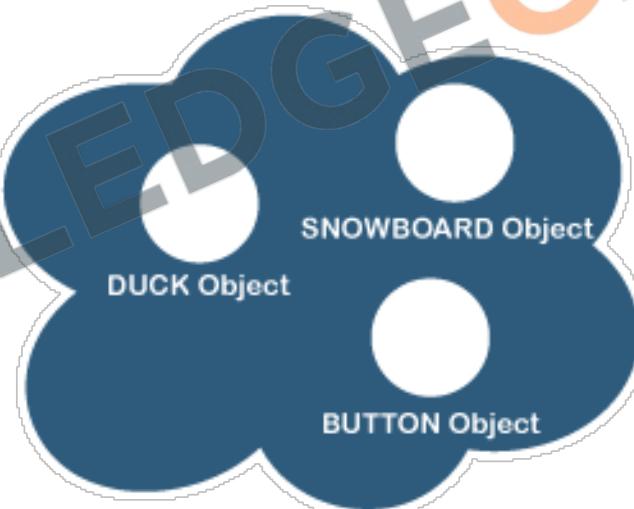
The Stack

Where method invocations
and local variables live



The Heap

Where ALL objects live



Stack Vs Heap

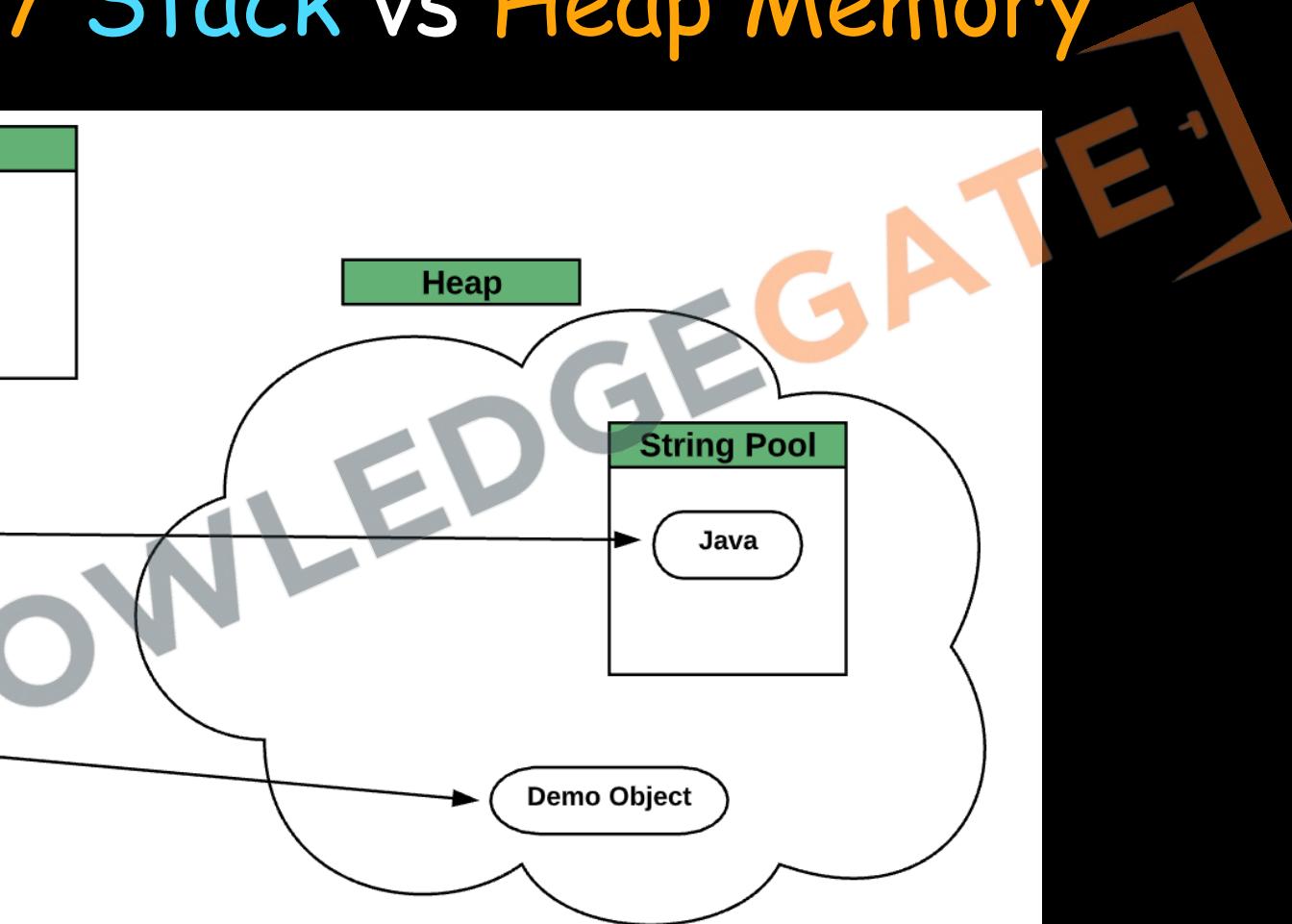


6.7 Stack vs Heap Memory

Code
{ int num = 50; String name = "Java"; Demo d = new Demo(); }

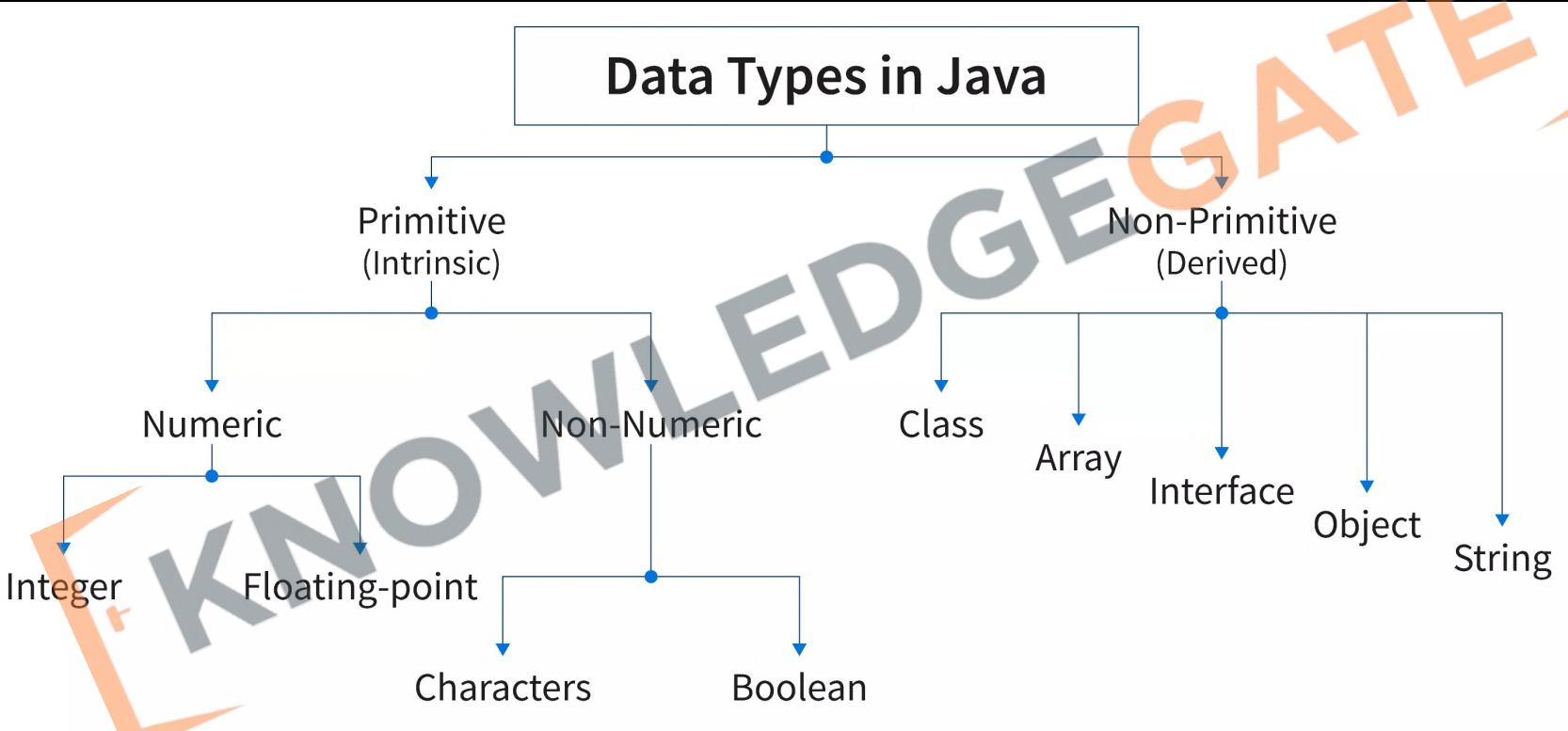
Stack
num = 50
name

d





6.8 Primitive vs Reference Types

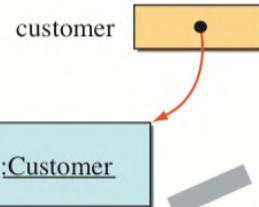


6.8 Primitive vs Reference Types

```
int number;  
number = 237;  
number = 35;
```

```
Customer customer;  
customer = new Customer();  
customer = new Customer();
```

number 237



1. **Memory:** Primitives store **actual values**; reference types **store addresses** to objects.
2. **Default Values:** Primitives have **specific defaults** like 0 or false; reference types default to null.
3. **Speed:** Access to primitives is **generally faster**.
4. **Storage Location:** Primitives are **stored in the stack**; reference types are **stored in the heap**.
5. **Comparison:** Primitives **compared by value**; reference types **compared by reference**.

6.9 Variable Scopes

Instance vs Local Variables

```
class Athlete {  
    public String  
    public double  
    public int  
        athleteName;  
        athleteSpeed;  
        athleteAge;  
  
    public Athlete( name, speed, age ){  
        this.athleteName = name;  
        this.athleteSpeed = speed;  
        this.athleteAge = age;  
    }  
    public void athleteRun(){  
        int speed = 100;  
        System.out.println("Athlete runs at"+ speed +"Km/hr");  
    }  
}
```

Instance Variables

Local Variables





6.9 Variable Scopes

global variable

```
public class MyScopeExample {  
    static String root = "I'm available  
    to all lines of code within my context";
```

local variable

```
public static void main(String[] args) {  
    String spy = "I'm a spy";  
    System.out.println(root); // Ok  
    System.out.println(spy); // Ok  
    System.out.println(anotherSpy); // Error  
}
```

local variable

```
public static void helpfulFunction() {  
    String anotherSpy = "I'm another spy";  
    System.out.println(root); // Ok  
    System.out.println(anotherSpy); // Ok  
    System.out.println(spy); // Error  
}
```

Local Scope

Global Scope

Local Scope

Multiple scopes



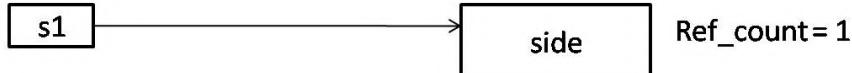
6.10 Garbage Collection & Finalize

1. **Automatic Process:** Garbage collection is **managed by the Java Virtual Machine (JVM)**, running in the background.
2. **Object Eligibility:** Objects that are **no longer reachable**, meaning no active references to them, are eligible for garbage collection.
3. **No Manual Control:** Unlike languages like C++, **Java developers cannot explicitly deallocate memory**. Garbage collection is automatic and non-deterministic.
4. **Generational Collection:** Java uses a generational garbage collection strategy, which divides memory into different regions (**young, old, and permanent generations**) based on object ages.
5. **Heap Memory:** Garbage collection **occurs in the heap memory**, where all Java objects reside.
6. **Performance Impact:** Garbage collection **can affect application performance**, particularly if it runs frequently or takes a long time to complete.

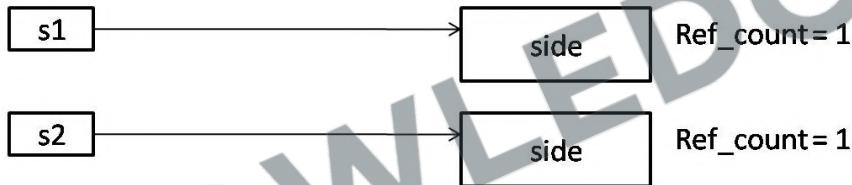


6.10 Garbage Collection & Finalize

`Square s1 = new Square();`

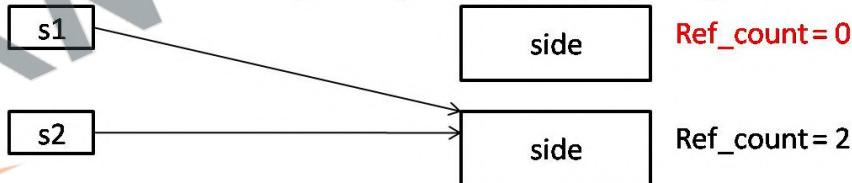


`Square s2 = new Square();`

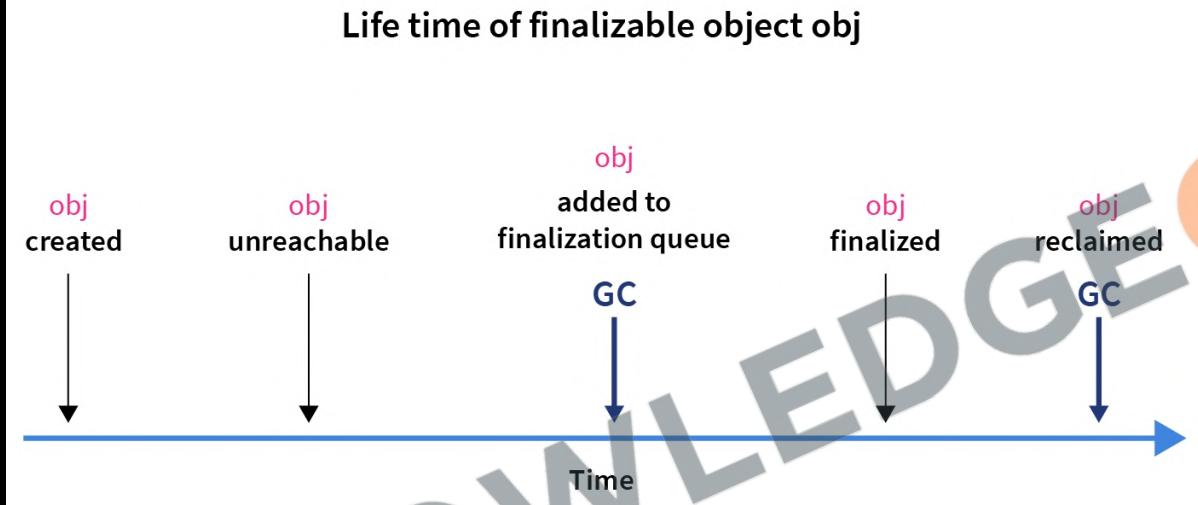


`s1 = s2;`

[This object is eligible for garbage collection]



6.10 Garbage Collection & Finalize



1. **Finalization:** Before an object is garbage collected, the `finalize()` method may be called, giving the object a chance to clean up resources. However, **it's not guaranteed to run**, and its usage is generally discouraged.
2. **Optimization:** Developers **can optimize the process indirectly through code practices**, like setting unnecessary object references to null.
3. **System.gc() Call:** While `System.gc()` suggests that the JVM performs garbage collection, it's not a guarantee.

Revision

1. Process vs Object Oriented
2. Instance Variables and Methods
3. Declaring and Using Objects
4. Class vs Object
5. This & Static Keyword
6. Constructors & Code Blocks
7. Stack vs Heap Memory
8. Primitive vs Reference Types
9. Variable Scopes
10. Garbage Collection & Finalize

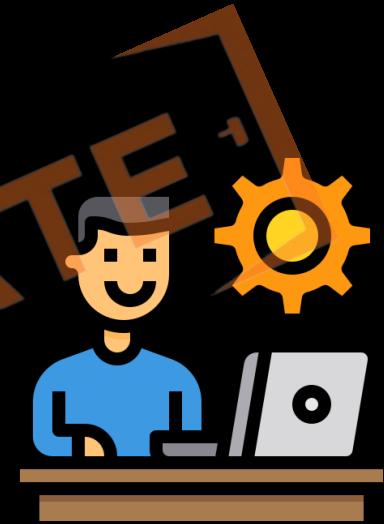


Practice Exercise

Classes, Objects

Answer in True/False:

1. Object-oriented programming is mainly concerned with data rather than logic.
2. Instance methods in Java can be called without creating an object of the class.
3. A class in Java is a template that can be used to create objects.
4. Static methods belong to the class and can be called without an object.
5. The new keyword is used to declare an array in Java.
6. Variables declared within a method are called instance variables.
7. In Java, you can access instance variables through static methods directly.
8. Every object in Java has its own unique set of instance variables.
9. The static keyword in Java means that a particular member belongs to a type itself, rather than to instances of that type.
10. Variable scope in Java is determined at runtime.



Practice Exercise

Classes, Objects

Answer in True/False:

1. Object-oriented programming is mainly concerned with data rather than logic. True
2. Instance methods in Java can be called without creating an object of the class. False
3. A class in Java is a template that can be used to create objects. True
4. Static methods belong to the class and can be called without an object. True
5. The new keyword is used to declare an array in Java. True
6. Variables declared within a method are called instance variables. False
7. In Java, you can access instance variables through static methods directly. False
8. Every object in Java has its own unique set of instance variables. True
9. The static keyword in Java means that a particular member belongs to a type itself, rather than to instances of that type. True
10. Variable scope in Java is determined at runtime. False



KG Coding

Some Other One shot Video Links:

- [Complete HTML](#)
- [Complete CSS](#)
- [Complete JavaScript](#)
- [Complete React and Redux](#)
- [One shot University Exam Series](#)



<http://www.kgcoding.in/>

Our  YouTube Channels

KG Coding Android App



[KG Coding](#)



[Knowledge GATE](#)



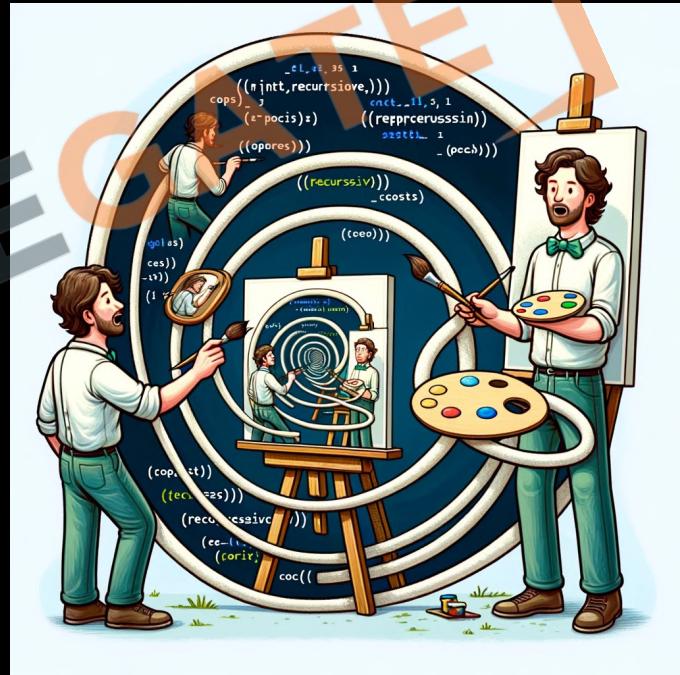
[KG Placement Prep](#)



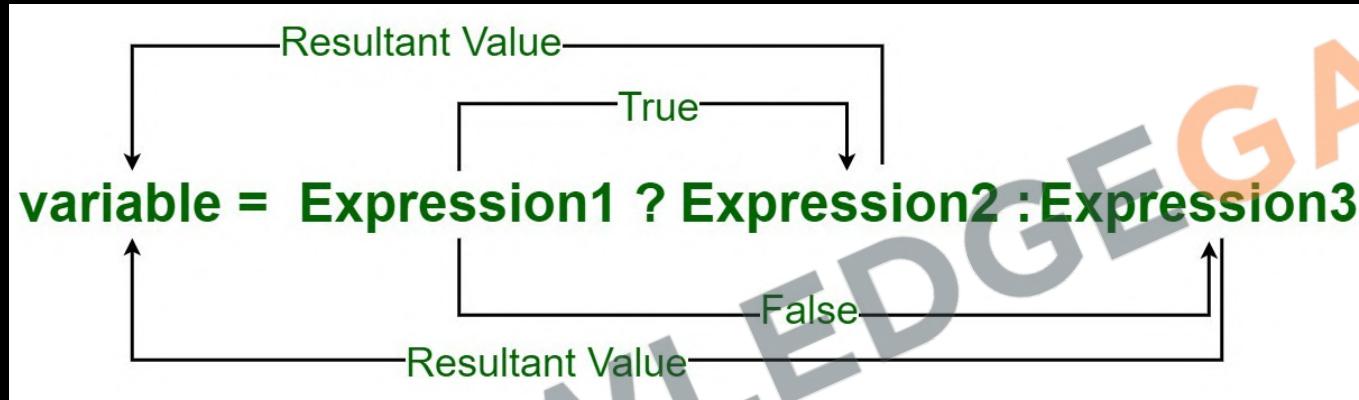
[Sanchit Socket](#)

7. Control Statements, Math & String

1. Ternary operator
2. Switch
3. Loops (Do-while, For, For each)
4. Using break & continue
5. Recursion
6. Random Numbers & Math class
7. Don't Learn Syntax
8. `toString` Method
9. String class
10. `StringBuffer` vs `StringBuilder`
11. Final keyword



7.1 Ternary operator



1. Syntax: `condition ? expression1 : expression2`
2. Condition: Boolean expression, evaluates to `true` or `false`.
3. Expressions: Both expressions **must return compatible types**.
4. Use Case: Suitable for **simple** conditional assignments.
5. Readability: Good for simple conditions, but can **reduce clarity** if overused.



7.2 Switch

```
switch (day) {  
    case 1: System.out.println("Monday");  
    break;  
    case 2: System.out.println("Tuesday");  
    break;  
    case 3: System.out.println("Wednesday");  
    break;  
    case 4: System.out.println("Thursday");  
    break;  
    case 5: System.out.println("Friday");  
    break;  
    case 6: System.out.println("Saturday");  
    break;  
    case 7: System.out.println("Sunday");  
    break;  
    default: System.out.println("Invalid day.");  
}
```

1. **Multiple Cases:** Handles **multiple values** for an expression efficiently.
2. **Supported Types:** Accepts **byte, short, char, int, String, enums**, and from Java 14, **long, float, double**.
3. **Case Labels:** Each case ends with a **colon (:)** and is followed by code.
4. **Break Statement:** Typically used to **prevent fall-through** between cases.
5. **Default Case:** Executes **if no case matches**; optional and doesn't require break.
6. **Type Safety:** Case label types must match the switch expression's type.



Java

7.2 Switch

```
String output = switch (day) {  
    case 1 -> "Monday";  
    case 2 -> "Tuesday";  
    case 3 -> "Wednesday";  
    case 4 -> "Thursday";  
    case 5 -> "Friday";  
    case 6 -> "Saturday";  
    case 7 -> "Sunday";  
    default -> "Invalid";  
};  
System.out.println(output);
```

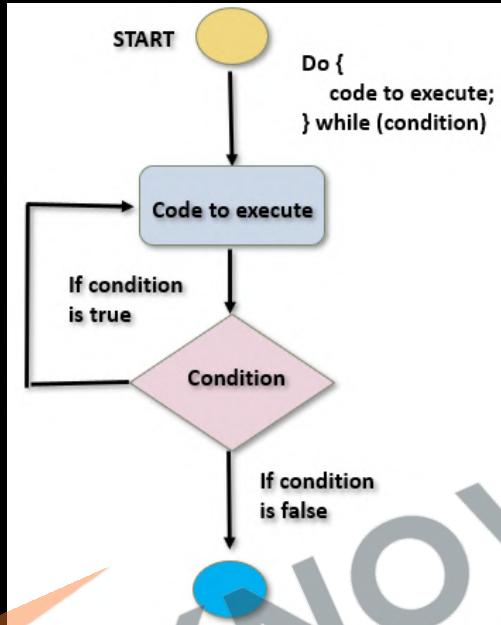
1. Enhanced Switch: Java 12 introduced enhancements like **yield** and **multiple constants** per case.
2. Switch Expression: From Java 14, **switch can return a value** using **yield**.

CHALLENGE

53. Create a program to **find the minimum of two numbers**.
54. Create a program to find if the **given number** is even or odd.
55. Create a program to **calculate the absolute value** of a given integer.
56. Create a program to Based on a student's score, categorize as "**High**", "**Moderate**", or "**Low**" using the ternary operator (e.g., High for scores > 80, Moderate for 50-80, Low for < 50).
57. Create a program to print the **month of the year** based on a **number (1-12)** input by the user.
58. Create a program to create a **simple calculator** that uses a **switch statement** to perform basic arithmetic operations like **addition, subtraction, multiplication, and division**.



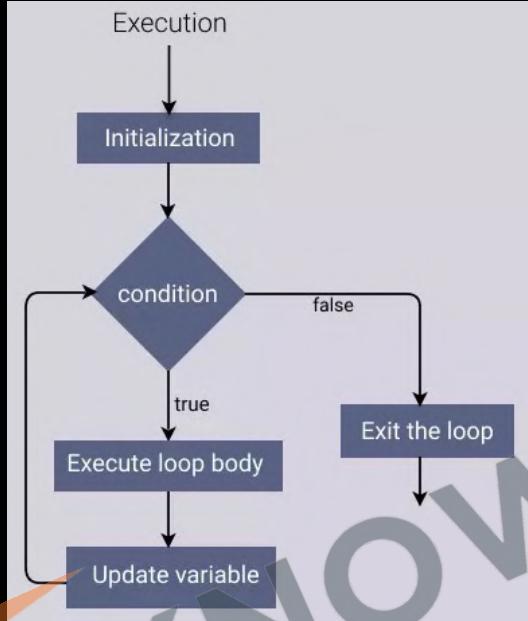
7.3 Loops (Do-while)



```
do {  
    // Body of the loop  
}  
while (condition);
```

1. Executes **block first**, then checks condition.
2. **Guaranteed** to run **at least one** iteration.
3. Unlike **while**, first iteration is **unconditional**.
4. **Don't forget** to update condition to **avoid infinite loops**.

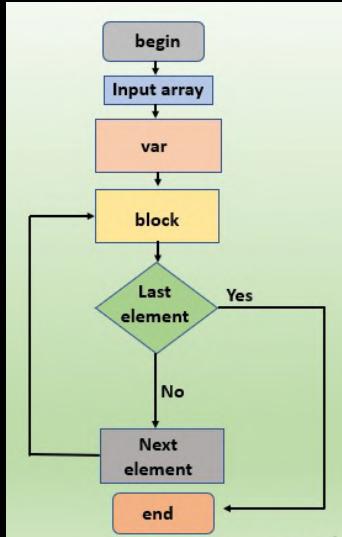
7.3 Loops (For)



```
for (initialisation; condition; update) {  
    // Body of the loop  
}
```

1. Standard loop for running code multiple times.
2. Generally preferred for counting iterations.

7.3 Loops (For each)



```
String[] names = new String[] {  
    "Ram", "Shyam", "Mohan", "Geeta",  
    "Sita", "Sohan"  
};  
for (String name: names) {  
    System.out.println(name);  
}
```

1. A method for **array iteration**, often preferred for readability.
2. Parameters: One for **item**, optional second for **index**.
3. Using **return** is similar to **continue** in traditional loops.
4. Not straightforward to **break** out of a forEach loop.
5. When you need to perform an action on each array element and don't need to break early.

7.4 Using break & continue

```
while (test condition)
{
    statement1;
    ....
    if (condition) true
        break;
    ....
    statement2;
}
```

out of the loop

top of the loop

```
while (test condition)
{
    statement1;
    ....
    if (condition) true
        continue;
    ....
    statement2;
}
```

1. Break lets you stop a **loop** early, or **break out of a loop**
2. Continue is used to skip one iteration or the current iteration
3. In **while loop** remember to do the **increment** manually before using **continue**



Java

7.5 Recursion

```
public static void main (String [ ] args) {
```

```
    recurse ( ) ——————
```

```
}
```

```
static void recurse ( ) { ←—————
```

```
    recurse ( ) ←—————
```

```
}
```

Recursive Call

Normal Method Call

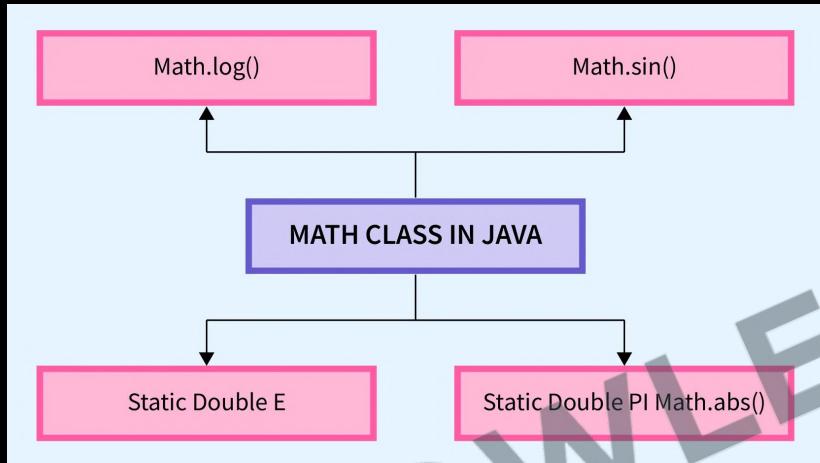
1. **Self-Calling Function:** Recursion is when a **function calls itself**.
2. **Base Case:** **Essential to stop recursion** and prevent **infinite loops**.
3. **Recursive Case:** The part where the function makes a recursive call.
4. **Stack Overflow Risk:** Excessive recursion **can cause stack overflow** errors.
5. **Problem Solving:** Ideal for **problems divisible into similar, smaller problems**.

CHALLENGE

59. Create a program using do-while to find password checker until a valid password is entered.
60. Create a program using do-while to implement a number guessing game.
61. Create a program using for loop multiplication table for a number.
62. Create a program using for to display if a number is prime or not.
63. Create a program using for-each to find the maximum value in an integer array.
64. Create a program using for-each to the occurrences of a specific element in an array.
65. Create a program using break to read inputs from the user in a loop and break the loop if a specific keyword (like "exit") is entered.
66. Create a program using continue to sum all positive numbers entered by the user; skip any negative numbers.
67. Create a program using continue to print only even numbers using continue for odd numbers.
68. Create a program using recursion to display the Fibonacci series upto a certain number.
69. Create a program using recursion to check if a string is a palindrome using recursion.



7.6 Random Numbers & Math class



Key Methods:

1. `abs()`: Absolute value.
2. `ceil()`: Rounds up.
3. `floor()`: Rounds down.
4. `round()`: Rounds to nearest integer.
5. `max()`, `min()`: Maximum and minimum of two numbers.
6. `pow()`: Power calculation.
7. `sqrt()`: Square root.
8. `random()`: Random number generation.
9. `exp()`, `log()`: Exponential and logarithmic functions.
10. Trigonometric functions: `sin()`, `cos()`, `tan()`.

1. **Static Class:** Math methods are **static** and accessed directly.
2. **Constants:** Includes **PI** and **E** for π and the base of natural logarithms.



7.7 Don't Learn Syntax

Google

ORACLE



1. **Google**: Quick answers to coding problems.
2. **Oracle**: In-depth guides and documentation.
<https://docs.oracle.com/>
3. **ChatGPT**: Real-time assistance for coding queries.
4. **Focus**: Understand concepts, not just syntax.



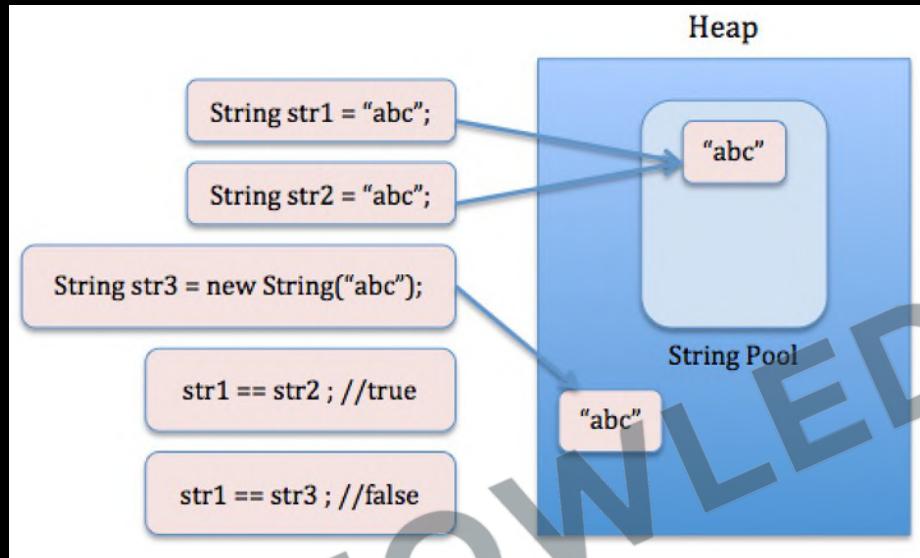
7.8 `toString` Method

```
@Override
public String toString() {
    return "Car{" +
        "noOfWheels=" + noOfWheels +
        ", color='" + color + '\'' +
        ", maxSpeed=" + maxSpeed +
        ", currentFuelInLiters=" + currentFuelInLiters +
        ", noOfSeats=" + noOfSeats +
        '}';
}
```

1. Function: `toString()` provides a string representation of an object.
2. Inheritance: It's **inherited from the Object class**.
3. Default Format: By default, **returns class name, "@", and hashCode**.
4. Overriding: Commonly overridden in custom classes for meaningful output.
5. Implicit Call: **Automatically called** in string concatenation.



7.9 String class



1. **Immutability:** Once created, a **String object's value cannot be changed**. Modifications create new String objects.
2. **String Pool:** Java maintains a **pool of strings for efficiency**. When a new string is created, it's checked against the pool for a match to reuse.
3. **Comparing:** `equals()` method for value comparison, `==` operator checks reference equality.



Java

```
"Car{" +  
"noOfWheels=" + noOfWheels +  
", color='\" + color + '\'' +  
", maxSpeed=" + maxSpeed +  
", currentFuelInLiters=" + currentFuelInLiters +  
", noOfSeats=" + noOfSeats +  
'}';
```

1. **Concatenation:** Strings can be **concatenated** using the **+** **operator**, but each concatenation **creates a new String**.
2. **Methods:** Provides methods like **length()**, **substring()**, **equals()**, **compareTo()**, **indexOf()**, for various operations.
3. **Memory:** Being immutable, strings can use more memory when frequently modified.



7.9 String class

Printf specifier	Data type
%s	String of text
%f	floating point value (float or double)
%e	Exponential, scientific notation of a float or double
%b	boolean true or false value
%c	Single character char
%d	Base 10 integer, such as a Java int, long, short or byte
%o	Octal number
%x	Hexadecimal number
%%	Percentage sign
%n	New line, aka carriage-return
%tY	Year to four digits
%tT	Time in format of HH:MM:SS (ie 21:46:30)

printf flag	Purpose
-	Aligns the formatted <i>printf</i> output to the left
+	The output includes a negative or positive sign
(Places negative numbers in parenthesis
0	The formatted <i>printf</i> output is zero padded
,	The formatted output includes grouping separators
<space>	A blank space adds a minus sign for negative numbers and a leading space when positive

% [flags] [width] [.precision] specifier-character



7.9 String class

Pattern	Data	<i>Printf</i> Output
'%s'	Java	'Java'
'%15s'	Java	'Java'
'%-15s'	Java	'Java' '
'%-15s'	Java	'JAVA' '



Java

7.9 String class

Pattern	Data	<i>Printf</i> output
'%d'	123,457,890	'123457890'
'%,15d'	123,457,890	' 123,457,890'
'%+,15d'	123457890	'+123,457,890'
'%_+,-,15d'	123457890	'+123,457,890 '
'%0,15d'	123457890	'0000123,457,890'

KNOWLEDGE GATE



7.10 StringBuffer vs StringBuilder

```
StringBuilder sentence = new StringBuilder("This is a sentence.");
sentence.append("Added word.");
System.out.println(sentence.toString()); //This is a sentence.Added word.
```

Parameter	String	StringBuilder	StringBuffer
mutability	immutable	mutable	mutable
Storage	String constant pool	heap	heap
Thread safety	Not used in the threaded environment as it is immutable	Not thread-safe so it is used in a single-threaded environment	Thread-safe so it is used in a multi-threaded environment
Speed	Comparably slowest	Comparably fastest	Faster than String but Slower than StringBuilder



Java

7.11 Final keyword

```
public class Main {  
    public final String name = "John";  
  
    public void setName(String name) {  
        this.name = name; // cannot reassign final variables  
    }  
}
```

1. **Variable:** When applied to a variable, **it becomes a constant**, meaning its **value cannot be changed** once initialized.
2. **Efficiency:** Using **final** can lead to **performance optimization**, as the compiler can make certain assumptions about final elements.
3. **Null Safety:** A final variable **must be initialized before the constructor completes**, reducing null pointer errors.
4. **Immutable Objects:** Helps in **creating immutable objects** in combination with private fields and no setter methods.

CHALLENGE

70. Define a **Student** class with fields like **name** and **age**, and use **toString** to print student details.
71. **Concatenate and Convert:** Take two strings, **concatenate** them, and convert the result to uppercase.
72. Calculate the area and **circumference** of a circle for a given radius using **Math.PI**
73. Simulate a dice roll using **Math.random()** and **display the outcome** (1 to 6).
74. Create a **number guessing game** where the **program selects a random number**, and the user has to guess it.
75. Take an array of **words** and **concatenate** them into a single string using **StringBuilder**.
76. Create an **object** with **final fields** and a **constructor** to initialize them.



Revision

1. Ternary operator
2. Switch
3. Loops (Do-while, For, For each)
4. Using break & continue
5. Recursion
6. Random Numbers & Math class
7. Don't Learn Syntax
8. `toString` Method
9. String class
10. `StringBuffer` vs `StringBuilder`
11. Final keyword





Practice Exercise

Control Statements, Math & String

Answer in True/False:

1. The ternary operator in Java can **replace** certain types of **if-else** statements.
2. A switch statement in Java can only **test for equality** with constants.
3. The do-while loop will **execute at least once** even if the condition is false.
4. A for-each loop in Java is used to **iterate over arrays**.
5. The break statement **skips** the current iteration.
6. The continue statement **exits the loop immediately**, regardless of the condition.
7. In Java, **recursion** is a process where a **method can call itself** directly.
8. Random numbers generated by **Math.random()** are inclusive of 0 and 1.
9. The String class is **mutable**, meaning it **can be changed** after it's created.
10. **StringBuilder** is faster than **StringBuffer** since it is not synchronized.
11. A final variable in **Java** can be assigned a **value once and only once**.

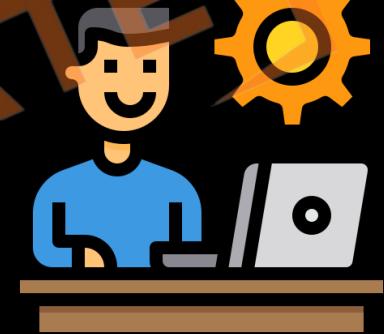


Practice Exercise

Control Statements, Math & String

Answer in True/False:

- | | |
|--|-------|
| 1. The ternary operator in Java can replace certain types of if-else statements. | True |
| 2. A switch statement in Java can only test for equality with constants. | True |
| 3. The do-while loop will execute at least once even if the condition is false. | True |
| 4. A for-each loop in Java is used to iterate over arrays . | True |
| 5. The break statement skips the current iteration. | False |
| 6. The continue statement exits the loop immediately , regardless of the condition. | False |
| 7. In Java, recursion is a process where a method can call itself directly. | True |
| 8. Random numbers generated by Math.random() are inclusive of 0 and 1. | False |
| 9. The String class is mutable , meaning it can be changed after it's created. | False |
| 10. StringBuilder is faster than StringBuffer since it is not synchronized. | True |
| 11. A final variable in Java can be assigned a value once and only once . | True |



KG Coding

Some Other One shot Video Links:

- [Complete HTML](#)
- [Complete CSS](#)
- [Complete JavaScript](#)
- [Complete React and Redux](#)
- [One shot University Exam Series](#)



<http://www.kgcoding.in/>

Our  YouTube Channels

KG Coding Android App



[KG Coding](#)



[Knowledge GATE](#)



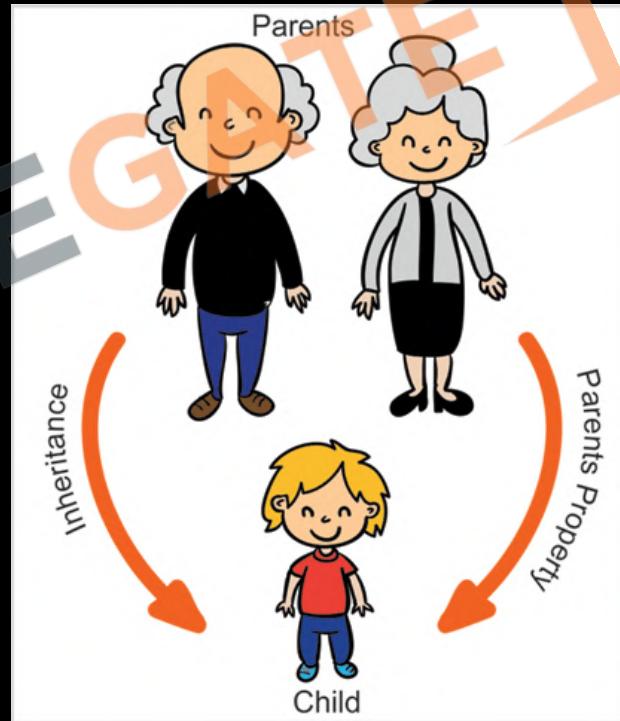
[KG Placement Prep](#)



[Sanchit Socket](#)

8 Encapsulation & Inheritance

1. Intro to OOPs Principle
2. What is Encapsulation
3. Import & Packages
4. Access Modifiers
5. Getter and Setter
6. What is Inheritance
7. Types of Inheritance
8. Object class
9. Equals and Hash Code
10. Nested and Inner Classes





Java

8.1 Intro to OOPs Principle

OOP Principles

Encapsulation

When an object only exposes the selected information.

Abstraction

Hides complex details to reduce complexity.

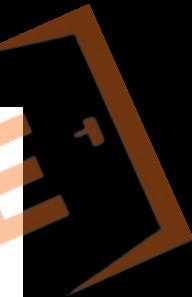
Inheritance

Entities can inherit attributes from other entities.

Polymorphism

Entities can have more than one form.

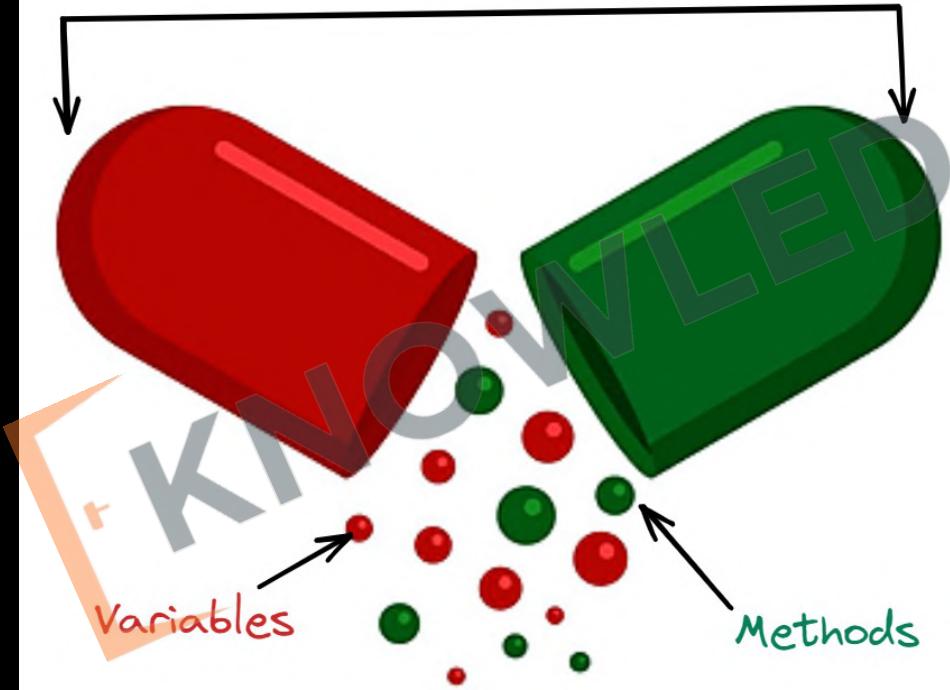
KNOWLEDGE GATE



8.2 What is Encapsulation

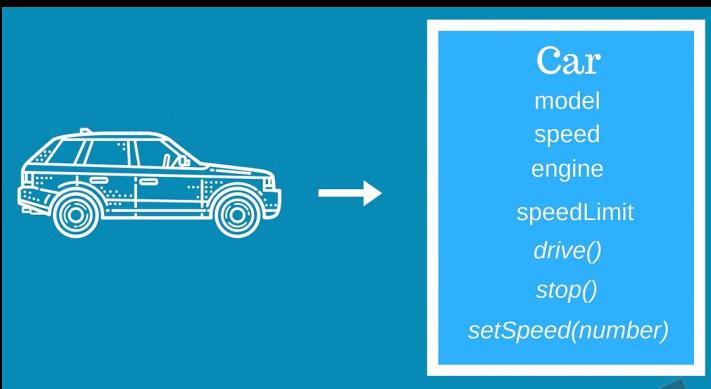
IN - CAPSULE - ation

class



ENCAPSULATION

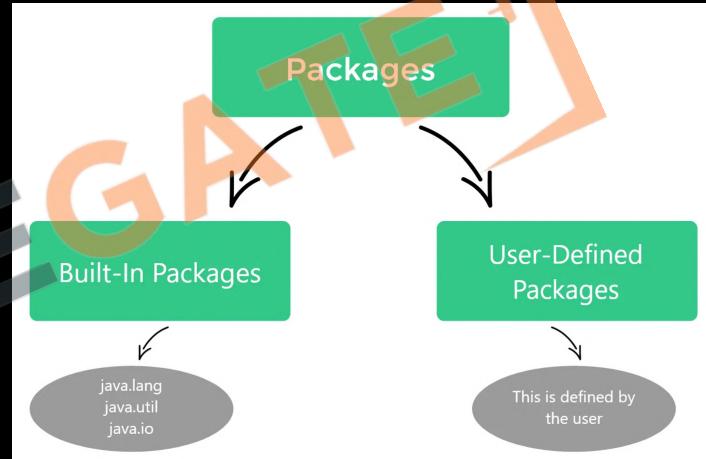
8.2 What is Encapsulation



1. **Data Hiding:** Encapsulation **hides internal data**, allowing access only through methods.
2. **Access Modifiers:** Uses **private, public, protected** to control access to class members.
3. **Getter/Setter:** Provides **public** methods for **controlled property access**.
4. **Maintains Integrity:** **Protects object state** from external interference.
5. **Enhances Modularity:** Keeps **classes separate** and reduces coupling.

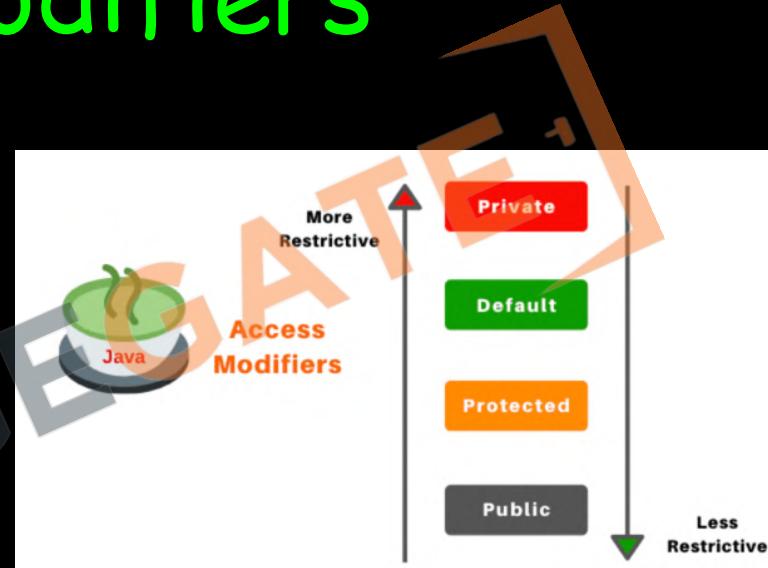
8.3 Import & Packages

1. **Package Definition:** A package in Java is a namespace that organizes classes and interfaces, preventing naming conflicts.
2. **Package Declaration:** Packages are declared at the beginning of a Java source file using the **package** keyword followed by the package name.
3. **Import Statement:** An **import statement** in Java is used to bring in classes or interfaces from other packages into the current file, making them accessible without using a fully qualified name.
4. **Types of Import:**
 - **Single-Type Import:** Imports a single class or interface from a package (e.g., `import java.util.List;`).
 - **On-Demand Import:** Imports all classes and interfaces from a package (e.g., `import java.util.*;`).
5. **Avoiding Collisions:** Packages help in avoiding name collisions by categorizing similar classes together.
6. **Built-in Packages:** Java comes with built-in packages like `java.lang` (automatically imported), `java.util`, `java.io`, etc.



8.4 Access Modifiers

1. **Types:** The four access modifiers are **public**, **protected**, **default** (no modifier), and **private**.
2. **Public:** The **public** modifier **allows access from any other class**. For classes, it means they can be accessed from any other package.
3. **Protected:** The **protected** modifier **allows access within the same package and subclasses**.
4. **Default:** If no access modifier is specified, it's the **default**, **allowing access only within the same package**.
5. **Private:** The **private** modifier **restricts access to the defining class only**.
6. **Class-Level Access:** **Only public and default** (no modifier) access modifiers are applicable for top-level classes.
7. **Member-Level Access:** Methods, constructors, and variables can use **all four access modifiers** to control visibility.





8.4 Access Modifiers

Access Specifiers in Java

		public	private	protected	default
Same Package	Class	YES	YES	YES	YES
	Sub class	YES	NO	YES	YES
	Non sub class	YES	NO	YES	YES
Different Package	Sub class	YES	NO	YES	NO
	Non sub class	YES	NO	NO	NO

8.5 Getter and Setter



AGGREGATE

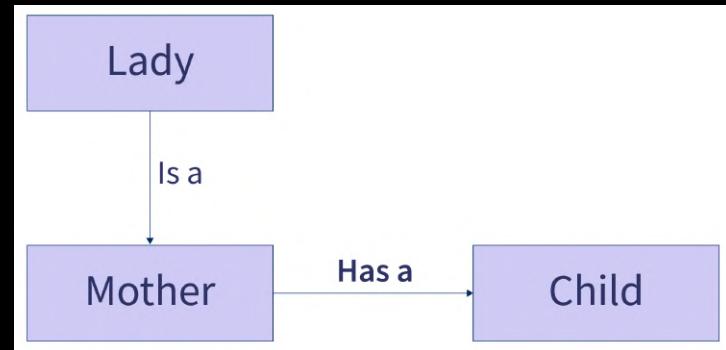
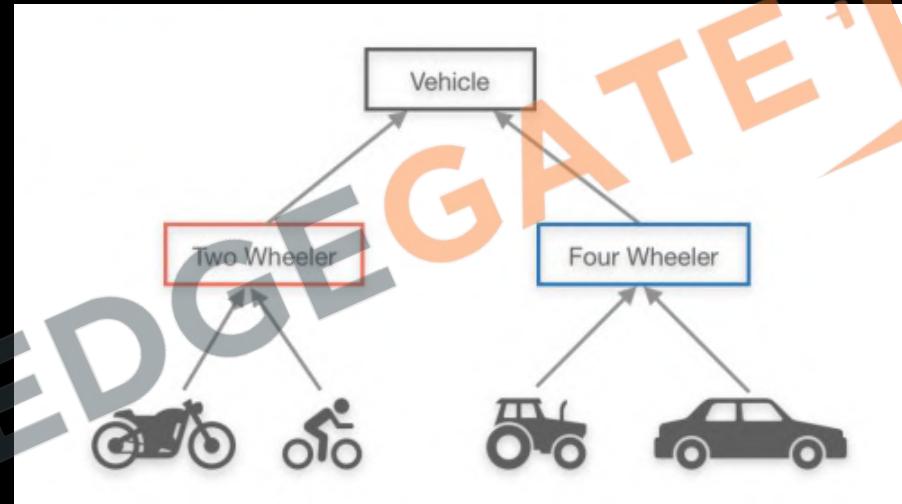
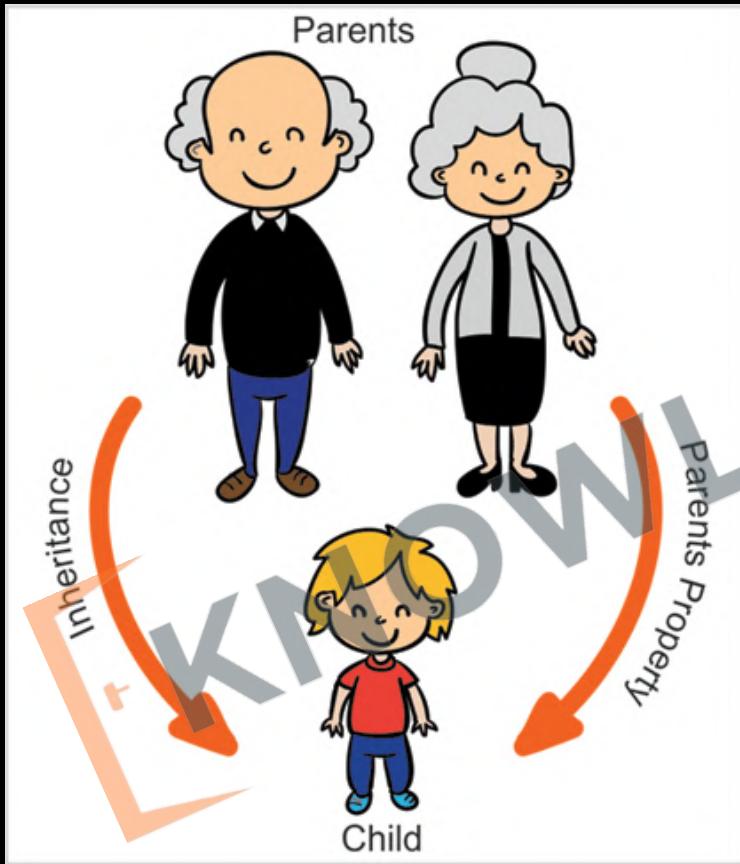
1. **Getters:** Retrieve private field values, typically named `get<FieldName>`.
2. **Setters:** Set or update private field values, usually named `set<FieldName>`.
3. **Control and Validation:** Offer controlled access and allow for validation logic.
4. **Encapsulation:** Facilitate read-only or write-only access to class fields.
5. **Flexibility:** Allow for internal changes without affecting external interfaces.

CHALLENGE

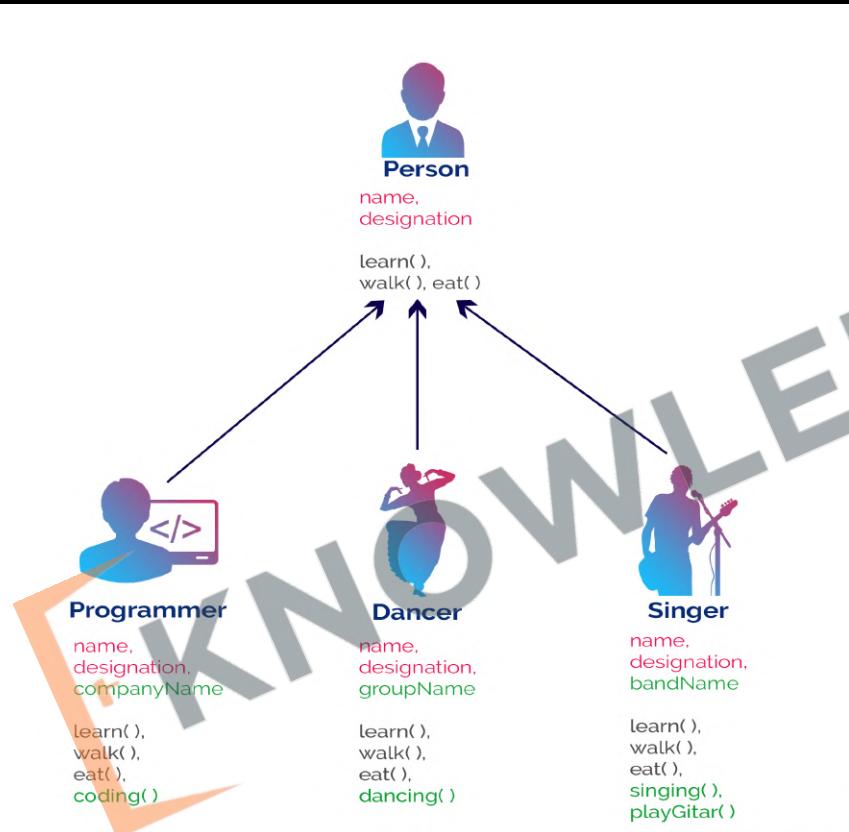
77. Create a simple application with at least two packages: `com.example.geometry` and `com.example.utils`. In the geometry package, define classes like Circle and Rectangle. In the utils package, create a Calculator class that can compute areas of these shapes.
78. Define a `BankAccount` class with private attributes like `accountNumber`, `accountHolderName`, and `balance`. Provide public methods to `deposit` and `withdraw` money, ensuring that these methods don't allow illegal operations like `withdraw`ing more money than the current `balance`.
79. Define a class `Employee` with private attributes (like `name`, `age`, and `salary`), public methods to get and set these attributes, and a package-private method to `displayEmployeeDetails`. Create another class in the same package to test access to the `displayEmployeeDetails` method.



8.6 What is Inheritance



8.6 What is Inheritance

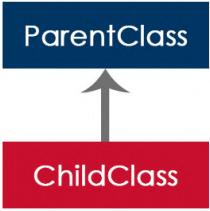


1. Inheritance allows a new class (**sub class**) to inherit features from an existing class (**super class**).
2. Code Reusability: It enables **sub classes** to use methods and variables of the **super class**, reducing code duplication.
3. Access Control: The **protected** access modifier is often used in inheritance to allow **sub class** access to **super class** members.

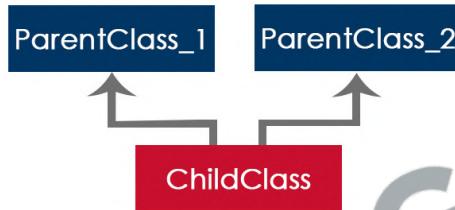


8.7 Types of Inheritance

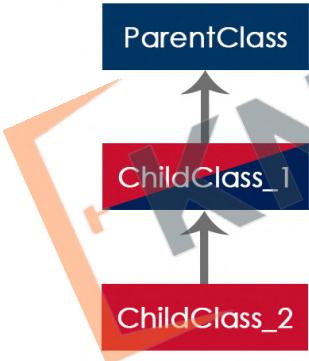
Simple Inheritance



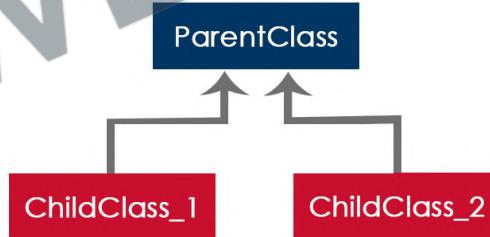
Multiple Inheritance



Multi Level Inheritance



Hierarchical Inheritance

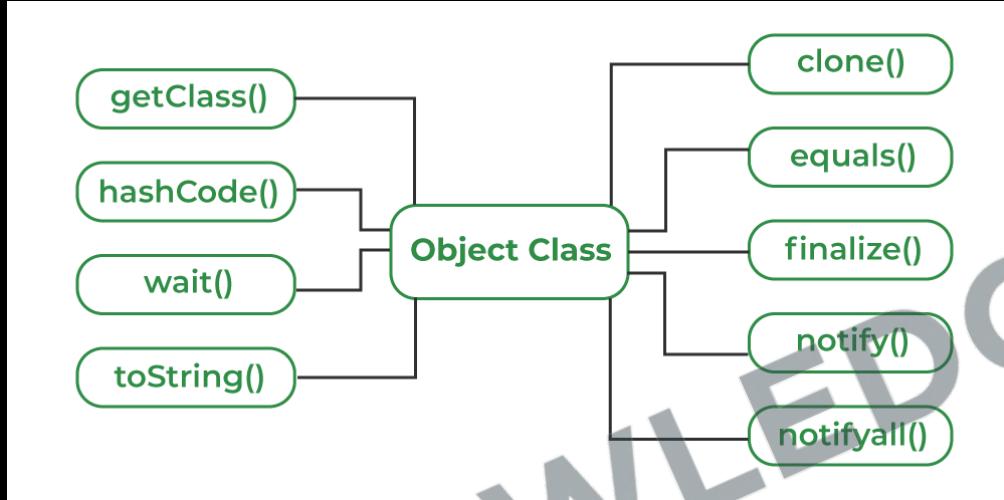


KNOWLEDGEABLE

1



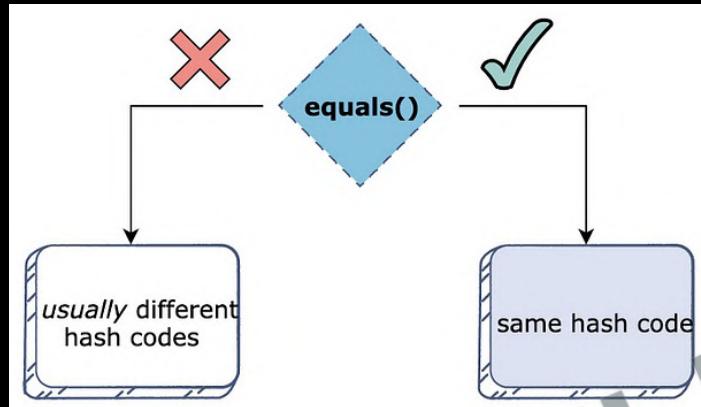
8.8 Object class



1. **Root Class:** The **Object** class is the **parent class** of all classes in Java, forming **the top of the class hierarchy**.
2. **Default Methods:** It provides fundamental methods like `equals()`, `hashCode()`, `toString()`, and `getClass()` that can be overridden by subclasses.
3. **String Representation:** The `toString()` method returns a string representation of the object, often overridden for more informative output.



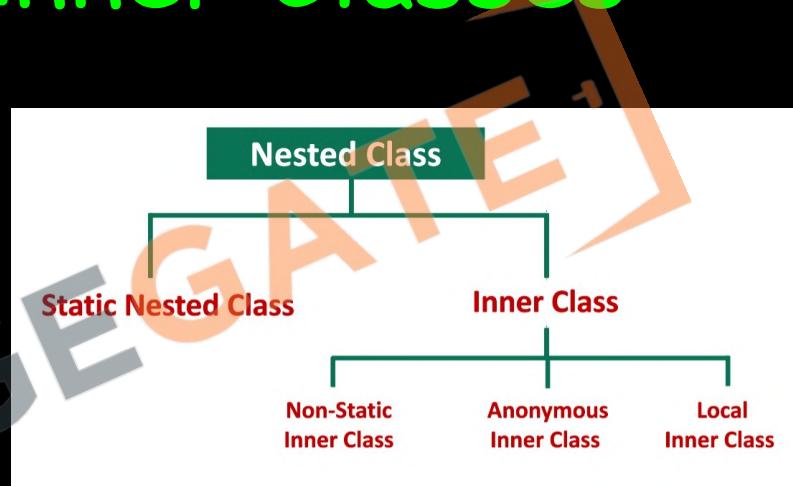
8.9 Equals and hashCode



1. **equals() Method:** Used for **logical equality** checks between objects. By default, it compares object references, but it's commonly overridden to compare object states.
2. **hashCode() Method:** Generates an **integer hash code representing an object**. It's crucial for the performance of hash-based collections like `HashMap`.
3. **Equals-HashCode Contract:** If two objects are **equal** according to `equals()`, they must have the **same hash code**. However, **two objects with the same hash code aren't necessarily equal**.
4. **Overriding Both:** If `equals()` is overridden, `hashCode()` should also be overridden to maintain consistency between these methods.

8.10 Nested and Inner Classes

1. **Nested Classes:** Classes defined within another class, divided into static (static nested classes) and non-static (inner classes).
2. **Static Nested Classes:** Act as static members of the outer class; can access outer class's static members but not its non-static members.
3. **Inner Classes:** Associated with an instance of the outer class; can access all members of the outer class, including private ones.
4. **Local and Anonymous Inner Classes:** Local inner classes are defined within a block (like a method) and are not visible outside it. Anonymous inner classes are nameless and used for single-use implementations.
5. **Use Cases:** Useful for logically grouping classes, improving encapsulation, and enhancing code readability.



CHALLENGE

80. Start with a base class `LibraryItem` that includes common attributes like `itemId`, `title`, and `author`, and methods like `checkout()` and `returnItem()`. Create subclasses such as `Book`, `Magazine`, and `DVD`, each inheriting from `LibraryItem`. Add unique attributes to each subclass, like `ISBN` for `Book`, `issueNumber` for `Magazine`, and `duration` for `DVD`.
81. Create a class `Person` with attributes `name` and `age`. Override `equals()` to compare `Person` objects based on their attributes. Override `hashCode()` consistent with the definition of `equals()`.
82. Create a class `ArrayOperations` with a static nested class `Statistics`. `Statistics` could have methods like `mean()`, `median()`, which operate on an array.



Revision

1. Intro to OOPs Principle
2. What is Encapsulation
3. Import & Packages
4. Access Modifiers
5. Getter and Setter
6. What is Inheritance
7. Types of Inheritance
8. Object class
9. Equals and Hash Code
10. Nested and Inner Classes



Practice Exercise

Encapsulation & Inheritance

Answer in True/False

1. Encapsulation is the OOP principle that ensures an object's data can be changed by any method, without restrictions.
2. The import statement in Java can be used to bring a single class or an entire package into visibility.
3. The private access modifier means that the member is only accessible within its own class.
4. Setters are used to retrieve the value of a private variable from outside the class.
5. Inheritance in OOP can be used to create a general class that defines traits to be inherited by more specific subclasses.
6. The Object class in Java has protected access by default.
7. If two objects are equal according to their equals() method, then their hashCode() methods must return different integers.
8. Inner classes are defined within the scope of a method and cannot exist independently of the method.
9. A static nested class can access the instance variables of its enclosing outer class.
10. Overriding the equals() method requires also overriding the hashCode() method to maintain consistency.



Practice Exercise

Encapsulation & Inheritance

Answer in True/False

1. Encapsulation is the OOP principle that ensures an object's data can be changed by any method, without restrictions. False
2. The import statement in Java can be used to bring a single class or an entire package into visibility. True
3. The private access modifier means that the member is only accessible within its own class. True
4. Setters are used to retrieve the value of a private variable from outside the class. False
5. Inheritance in OOP can be used to create a general class that defines traits to be inherited by more specific subclasses. True
6. The Object class in Java has protected access by default. False
7. If two objects are equal according to their equals() method, then their hashCode() methods must return different integers. False
8. Inner classes are defined within the scope of a method and cannot exist independently of the method. False
9. A static nested class can access the instance variables of its enclosing outer class. False
10. Overriding the equals() method requires also overriding the hashCode() method to maintain consistency. True



KG Coding

Some Other One shot Video Links:

- [Complete HTML](#)
- [Complete CSS](#)
- [Complete JavaScript](#)
- [Complete React and Redux](#)
- [One shot University Exam Series](#)



<http://www.kgcoding.in/>

Our  YouTube Channels

KG Coding Android App



[KG Coding](#)



[Knowledge GATE](#)



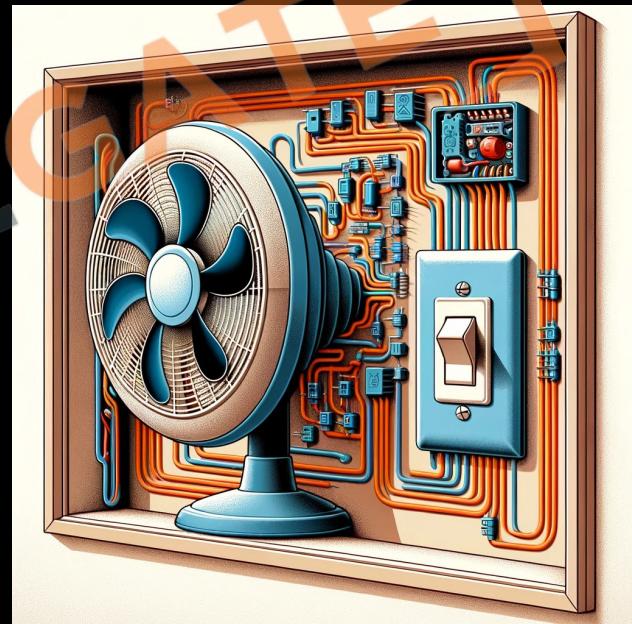
[KG Placement Prep](#)



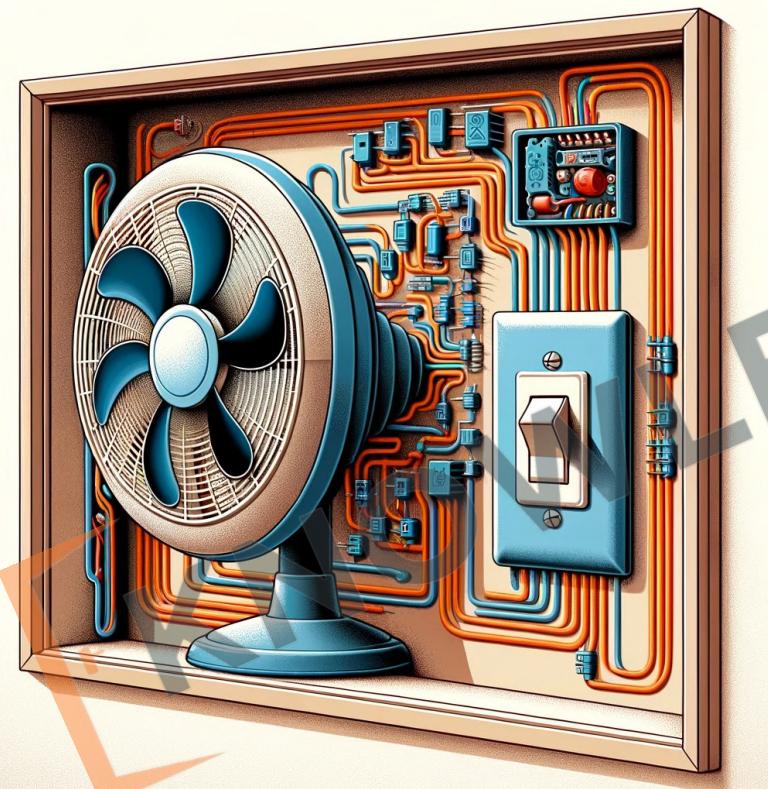
[Sanchit Socket](#)

9 Abstraction and Polymorphism

1. What is **Abstraction**
2. **Abstract Keyword**
3. **Interfaces**
4. What is **Polymorphism**
5. **References and Objects**
6. **Method / Constructor Overloading**
7. **Super Keyword**
8. **Method / Constructor Overriding**
9. **Final keyword revisited**
10. **Pass by Value vs Pass by reference.**

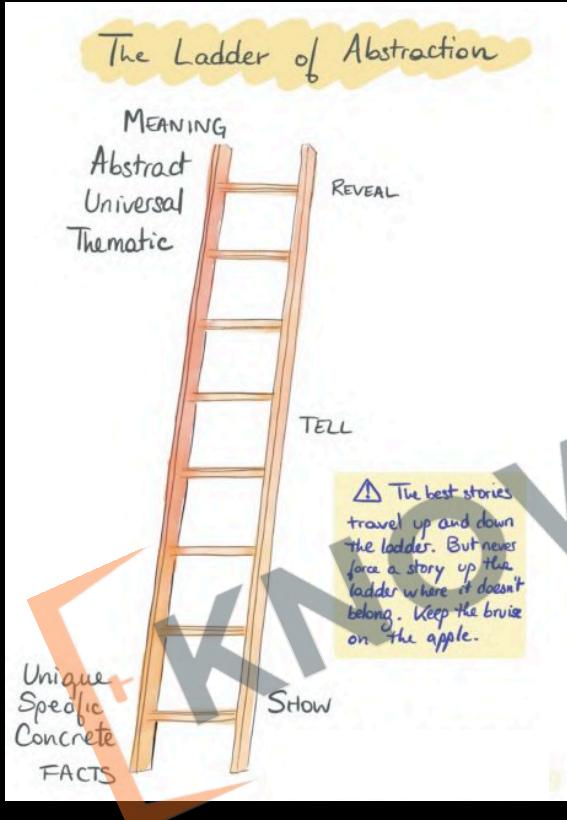


9.1 What is Abstraction



1. Core Principle: Abstraction **hides complex implementation details**, focusing only on essential features.
2. Focus on Functionality: Emphasizes **what an object does, not how it does it**, through clear interfaces.

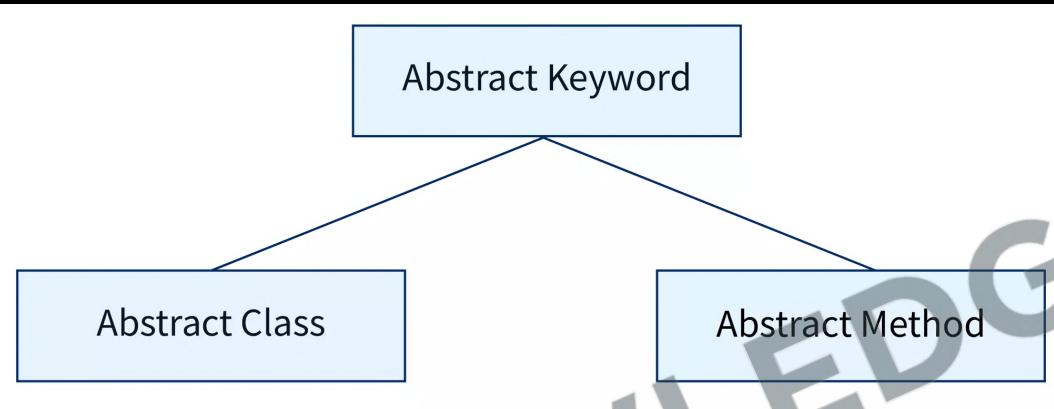
9.1 What is Abstraction



1. **Simplifies Complexity:** It reduces complexity by showing only relevant **information** in class design.
2. **Real-World Modelling:** Abstraction allows creating objects that represent real-life entities with key attributes and behaviours.



9.2 Abstract Keyword



1. **Abstract Class:** Used to declare **non-instantiable abstract classes** that serve as base classes.
2. **Abstract Method:** Defines **methods without implementations**, requiring subclasses to provide specific functionality.
3. **Mandatory Implementation:** Subclasses **must implement all abstract methods** of an abstract class.
4. **Design Flexibility:** Allows for flexible class design by **defining a contract** for subclasses.